From Curry to Haskell: Paths to Abstraction in Programming Languages

Noname manuscript No.
(will be inserted by the editor)

From Curry to Haskell*

Paths to abstraction in programming languages

Felice Cardone

the date of receipt and acceptance should be inserted later

Abstract We expose some basic elements of a style of programming supported by functional languages like Haskell by relating them to a coherent set of notions and techniques from Curry's work in combinatory logic and formal systems, and their algebraic and categorical interpretations. Our account takes the form of a commentary to a simple fragment of Haskell code attempting to isolate the conceptual sources of the linguistic abstractions involved.

1 Introduction

"From the moment that von Neumann first suggested that the instructions and data share the same storage in a machine there has been a growing realization that general purpose digital computers are linguistic mechanisms" (Gorn 1959). This early statement by Saul Gorn is the conclusion of an argument exploiting both the universality of computers, allowing to interchange software and hardware when we "recognize that a program for a general purpose machine turns it into a special purpose machine", and "the equivalence of general purpose machines with a vaguely recognizable 'universal command language'."

However, the design of such a language must take into account the fact that a program has not merely an operational significance. The way a piece of software is written connects it and the problem it is designed to solve to a conceptual background that determines to a large extent the software architecture and the linguistic abstractions supporting it.

Logic, in a wide sense, has often been one main source of these insights: think for example of the set-theoretical basis of the data types that we find already in the context of imperative programming, such as record, enumeration

* Published in Philosophy & Technology, 2020. The final publication is available at link.springer.com, https://link.springer.com/article/10.1007/s13347-019-00385-4

Dipartimento di Informatica, Università di Torino. E-mail: felice.cardone@unito.it

and array types (Hoare 1972), and its far-reaching extensions to universes of constructive objects leading to intuitionistic type theories (Martin-Löf 1982) or impredicative theories of constructions (Coquand and Huet 1988).

In this paper we would like to propose a case study of how a small set of notions arising from logic, algebra and category theory, has provided syntactical and semantical guidelines to the formation of a currently widespread programming style. The main characters of our account will be the combinatory logic developed by Haskell B. Curry in the late 1920s motivated by problems from the foundations of mathematics, and the programming language Haskell (Hudak et al. 2007). Of the first, we shall be especially interested in the aspects in which we believe it has offered the most significant contributions to the foundations of programming:

- The abstract nature of the formal systems of combinatory logic (Curry and Feys 1958; Curry et al. 1972): since the early stages of development of his approach to logic and mathematics, Curry has intentionally put much effort towards achieving abstraction as a substantial part of his formalistic foundational program (Curry 1941; 1950; 1951; 1953). In particular, an abstract formal system never displays a concrete representation of its formal objects, the *obs*, in Curry's neologism.
- The use of functional application as the only primitive way of combining the objects of combinatory systems, leading to a general notion of applicative syntax extending to fields outside logic, for example the grammatical structure of natural languages (Curry 1961). Applicative syntax is also one of the key linguistic advances of Landin's path-breaking paper on the mechanical evaluation of expressions (1964), and a permanent feature of most functional programming languages.
- The interest of Curry for the structural properties of formal objects, that makes several of his purely syntactical results easily expressible in algebraic terms (Curry 1952), as we shall see below. This has inspired a systematic use of structure definitions (Landin 1964) and the associated techniques of structural induction (Burstall 1969).
- The early use by Curry of Gentzen-style natural deduction and sequent calculus, and related proof-theoretical techniques like the *inversion principle* of Paul Lorenzen (Lorenzen 1955). These fit nicely within Curry's philosophical outlook and, from the programming point of view, also provide a way of relating recursion over data types to the inductive generation of their elements.

In our account the main emphasis will be on the conceptual links between notions, and not on a detailed historical reconstruction of causal dependences from Curry's work that are in general rather episodic, even though sometimes substantial, like those that can be found in the seminal works of Landin (1964) and Burstall (1969) mentioned above, two milestones in the development of

functional programming.[1] Our aim is to document the existence of a coherent body of notions and techniques that have provided the ground from which have grown the abstractions that find linguistic support in languages like Haskell. This coherence is perhaps the only element of continuity in a plot where Curry and Haskell are two among the many actors on stage. We take the contribution of this paper to consist in gathering evidence supporting this claim of coherence.

Our story will be based on a sample of Haskell code from (Bird 1998), shown in Figure 1, defining sum, product and exponentiation over elements of an algebraic data type `Nat` of natural numbers. This will allow us to illustrate concretely a programming style centered around mathematical structures with a formally defined semantics and a natural intuitive content that arises from the background that we are going to explore.

```
data Nat = Zero | Succ Nat

foldn :: (t -> t) -> t -> Nat -> t
foldn h c Zero = c
foldn h c (Succ n) = h (foldn h c n)

add m n = foldn Succ m n
mult m n = foldn (add m) Zero n
pow m n = foldn (mult m) (Succ Zero) n
```

Fig. 1 Addition, multiplication and exponentiation as `fold`s.

2 Formal systems, syntax and data types

Formal systems are a far reaching generalization of the way of generating natural numbers by iterated applications of the successor function to the initial element 0. They are also an abstract model of the way theorems are derived from axioms by means of inference rules in a system of formal logic. Their study as mathematical objects has roots in the work of Post in the 1920s (Post 1943; De Mol 2006). Curry, apparently independently of Post's work, based on a notion of formal system his formalist position in the philosophy of mathematics (Curry 1941; 1950; 1951; 1953; Seldin 1975; Meyer 1987). Formal systems, in the sense of Curry and Feys (1958) and Curry (1963), will be the starting point of our search for conceptual roots of the abstractions exploited in the code in Figure 1.

[1] Also the MIT course notes on programming linguistics (Wozencraft and Evans 1971), strongly influenced by lectures held at MIT by Landin and Strachey, use some of Curry's terminology, speaking of 'obs' and their 'representations'.

2.1 From formal systems to abstract syntax

The purpose of a formal system is that of specifying an *inductive class* $\mathfrak{X}$ which, as usual, consists of *initial elements*, called the *basis* $\mathfrak{B}$ of $\mathfrak{X}$, and a class of *modes of combination* whose elements μ have a *degree*, the number of elements of $\mathfrak{X}$ (the *arguments* of μ) to which μ is applied to produce an element of $\mathfrak{X}$ (Curry 1963, §2A5,2C3). As an example, that we also use to introduce the basic terminology, consider the problem of generating the inductive class of natural numbers. We may do this in two ways:

1. in the first approach we have an initial element 0 and one mode of combination s with one argument. The elements of the inductive class generated by these data are the natural numbers.
2. In the second approach, we have as initial element the statement $N(0)$, and one mode of combination that given a statement $N(x)$ produces a statement $N(s(x))$. The elements of this inductive class can be interpreted as a deductive system where the initial element is an axiom and the single mode of combination is a deductive rule.

Both approaches are instances of the general notion of formal system introduced by Curry, where we have:

- the specification of the *formal objects* (the *obs*, as Curry used to call them) by means of *operations* and *atoms*, here s and 0,
- the specification of the *elementary statements* by giving the *basic predicates*, in this case a unary predicate N.

Finally, we also need

- the specification of *deductive rules* for deriving certain statements as theorems starting from *axioms*. In the example the statement $N(0)$ is taken as axiom, with rules

$$\frac{}{N(0)}\ \text{(zero)} \qquad \frac{N(x)}{N(s(x))}\ \text{(successor)} \tag{1}$$

A distinctive feature of Curry's approach is that a formal system is presented in an extension of the natural language which is being used for communication, that Curry calls the *U-language*: a formal system is an activity that is carried out in the U-language (Curry et al. 1972, §11A, p. 4). The *obs* of the system are the results of the inductive process of repeatedly applying the operations, starting from the atoms, to previously obtained obs. The important thing to notice is that, in this approach, the obs are never displayed explicitly, but only through the notations used for naming them in the U-language. These notations define a *presentation* of the obs. For example, in a system with an operation ω of degree n the result of the application of ω to obs $a_1, \ldots, a_n$ may be presented by '$\omega(\dot{a}_1, \ldots, \dot{a}_n)$' or by the Łukasiewicz presentation '$\omega\dot{a}_1 \cdots \dot{a}_n$', where dotted letters are the respective presentations of the obs. Curry speaks of an ob as the objectification of its construction process, which is assumed to be

uniquely associated to the ob; presentations have to guarantee this uniqueness of parsing. Of course we may associate a *representation* to the system by mapping each ob to a unique thing, which may be for example a combination of symbols, or a manufactured thing (Curry and Feys 1958, §1C2), like one of Calder's mobiles. Formal systems are *abstract* when no such representation is specified. It is this way of understanding abstraction that will become central for programming languages after (McCarthy 1963): "The form of syntax we shall now describe [...] is abstract in that it is independent of the notation used to represent, say sums, but only affirms that they can be recognized and taken apart" (cit., §12, p. 26).

From the above sketch of the basic ingredients of a formal system in the writings of Curry, we can easily see the relevance of this notion for the early speculations on the syntax of programming languages, in the form of the definition of *structures*: "A structure is obtained by taking a sequence of objects (atoms or structures) and combining them with a construction operation. Thus each structure is built up from atoms by using a finite number of construction operations" (Burstall 1969, p. 42). Structures correspond to the obs of Curry; the obs of a formal system are subject to the same analysis and synthesis operations as the corresponding structures are.

2.2 From abstract syntax to the mechanical evaluation of expressions

The uniqueness of construction required of obs, and the corresponding observation that "each structure has a unique set of components" (Burstall 1969) leads to recognizing their common nature as elements of word algebras of type Ω, namely algebras $W_\Omega(X)$ whose elements are obs built from atoms and operations in Ω possibly using elements of the arbitrary set X as new atoms.[2]

A concrete representation (in the sense of Curry mentioned above) of the elements of a word algebra is obtained by means of reverse Polish notation (Cohn 1965, §III.2), where an ob $\omega(a_1, \ldots, a_n)$ is written as the concatenation of the representations of the obs $a_1, \ldots, a_n$ followed by ω. It is well-known that each such representation can be parsed uniquely even without the use of parentheses, by the following algorithm:

1. let the rank of an atom be 1, the rank of an operation of degree n be $-n+1$.
2. scan from left to right the sequence of symbols adding their ranks: if, and only if, the partial sums obtained in this process are always positive and the final sum is 1, the sequence represents an ob.

The history of this result has been told in part in the textbook of Rosenbloom (1950) who cites the early proofs by Menger (1930), Schröter (1943), Gerneth

[2] Curry (1952; 1958) observed the analogy between formal systems and abstract algebras, pointing out also their main differences, namely the fact that in an algebra "the elements are conceived as existing beforehand" , where in a formal system "what is given beforehand is not a set of elements but the atoms and operations, and the obs are generated from them" (Curry and Feys 1958, §1B1).

(1948) and an unpublished one of Philip Hall (around 1950). The other part of the history is told in (Bauer 1990; 2002) and concerns the early attempts to mechanize the evaluation of expressions, notably (Burks et al. 1954). In fact it is possible to read a reverse Polish expression of the form, say, $5\ 17\ +\ 2\ *$ as a sequential program that evaluates the expression by means of a stack structure, where

- a numeral is interpreted as the instruction to push the corresponding numerical value onto the stack, and
- the operation symbols as the instruction to perform the corresponding operation on the two topmost numbers on the stack, replacing them by the result.

The transitions of the stack in the evaluation of the above expression would be therefore the following, where the the stack grows leftwards:

$$\langle\ \ \rangle \xrightarrow{\text{push}} \langle 5\rangle \xrightarrow{\text{push}} \langle 17\ 5\rangle \xrightarrow{+} \langle 22\rangle \xrightarrow{\text{push}} \langle 2\ 22\rangle \xrightarrow{*} \langle 44\rangle.$$

From the point of view of programming, reverse Polish notation allows to regard expressions as *programs* for computing their values. This observation is constitutive of the very idea functional programming, and is the germ of the mechanical evaluation of expressions envisaged by Landin (1964), leading to his design of the SECD-machine.

An important property of this representation of the elements of $W_\Omega(X)$ is that it provides a direct way of proving that, for any structure A of type Ω, and any association of elements of (the carrier of) A to the elements of X, there is a unique mapping $W_\Omega(X) \longrightarrow A$ preserving the operations in Ω (a *Ω-homomorphisms*) and extending the given association. This opens the way to a connection of formal syntax with abstract algebraic notions that will be an important theme throughout the paper.

2.3 Applicative syntax and grammatical categories

The formalization of combinatory logic was based on the observation (made by Schönfinkel in the early 1920s) that functions of several arguments can be transformed into functions with one argument, possibly returning functions as results. This transformation is now well-known as *currying* (Bird 1998). Building on this remark, Curry devised a general transformation that turns a general formal system into an *applicative* system, where the only operation is a binary operation of application of two obs a and b. This notational device is exploited systematically in the code displayed in Figure 1.

Applicative syntax allowed Curry (1934) to introduce a uniform notation for expressing type distinctions in his systems of illative combinatory logic (Curry and Feys 1958, Ch. 9), the *theory of functionality*, that was later extended to a formulation of the grammatical structure of artificial and natural languages (Curry 1961). In the latter case we can assume two basic *grammatical categories* n for nouns and s for sentences, and build new grammatical

categories by the recursive specification that whenever α, β are grammatical categories, then $\mathbf{F}\alpha\beta$ is the grammatical category of phrases (*functors*) that form a phrase of category β when they are applied to a phrase of category α. For example we have then that the category of adjectives is $\mathbf{F}nn$ because, for example, the adjective 'good' can be prefixed to the noun 'man' to form the noun 'good man'. In the language of mathematics this category includes, for example, the square function. Similarly, a phrase like '− fly' has category $\mathbf{F}ns$ because it forms a sentence when applied to (i.e., the dash is replaced by) a noun, like in 'birds fly'. The grammatical rules for operators from logic and mathematics can be reformulated in these terms, and indeed Curry claimed that nouns, sentences and functors are "the fundamental categories of any conceivable mathematical language" (Curry 1961, p. 61). So, for example, unrestricted quantifiers are given category $\mathbf{F}(\mathbf{F}ns)s$, whereas derivation of polynomials is given category $\mathbf{F}(\mathbf{F}nn)(\mathbf{F}nn)$, and the summation and min, max operators have category $\mathbf{F}(\mathbf{F}nn)n$ (Curry and Feys 1958, §§8C,8S2).[3]

Furthermore applicative syntax, together with a flexible type discipline based on Curry's theory of functionality and type inference by means of the Hindley-Milner algorithm (Hindley 1969; Milner 1978), supports a peculiar kind of abstraction in functional languages. By permuting arguments, we can define from the functional `foldn` shown in Figure 1 a new functional

```
it :: Nat -> (a -> a) -> (a -> a)
```

that can also be used to represent the general idea that natural number n is an *iterator*, as in the definition of the common abbreviation f^n for $\underbrace{f \circ \cdots \circ f}_{n \text{ times}}$, the n-fold composition of any function of type `a -> a` with itself.

Applicative syntax was systematically exploited in the seminal paper by Landin (1964), who observes that "many symbolic expressions can be characterized by their 'operator/operand' structure [...] or 'applicative' structure" (p. 308). Such *applicative expressions* (AEs), also including λ-abstractions, are another example of abstract syntactical objects of the kind used in the ob systems of Curry (the works of Curry and McCarthy have been quite influential on the ideas of this classic paper by Landin): "there are many ways in which we can write the same AE, differing in layout, use of brackets and use of infixed as opposed to prefixed operators. However, they are all written representations of the *same* AE" (ibid., p. 314).

3 Free structures

The data type declaration

[3] As an aside, we point out that Giovanni Vailati, a collaborator of Peano, had already studied the language of algebra and its grammar, in 'La grammatica dell'algebra' (Rivista di Psicologia Applicata, **4**, 1908), to which Peano replied more than twenty years later with his 'Algebra de Grammatica', Schola et Vita, vol. V (1930) pp. 323–336, where he outlines an algebraic approach to grammar based on the categories of verb, noun and adjective that is strongly reminiscent of the more successful subsequent attempts by Ajdukiewicz, Bar Hillel and especially Lambek.

```
data Nat = Zero | Succ Nat
```

makes available the constant `Zero` and the unary *constructor* `Succ` for building elements of type `Nat` intended to represent the natural numbers. Then we have expressions

`Zero`, `Succ Zero`, `Succ (Succ Zero)`, ...

all of type `Nat`. The idea of algebraic data types like `Nat` shows up in a rather mature form in the fundamental paper by Burstall (1969), influenced by Landin (1964). In the latter paper lists are defined (on p. 312) as follows, by what Landin calls a *structure definition*:

> A *list* is either null or else has a *head* (h) and a *tail* (t) which is a *list*.

This is an *analytic* definition (McCarthy 1963, §12): it tells how to take a structure apart, rather than how to put it together. Burstall (1969) introduced lists through a *synthetic* definition, by privileging constructors *cons* and *nil* over destructors *head* and *tail*:

> A *list* is either a *cons* or *nil*. A *cons* has an *atom* and a *list*. A *nil* has no components.

Such synthetic definitions take the form of conjunctive clauses specifying how the constructors operate on components; these constitute the cases of an overall disjunction. A structure definition similar to that used for lists can also be used in defining applicative expressions.

While the elements of an algebraic data type are intended to be built inductively, and therefore their construction process can be described faithfully by the derivations of a formal system, the intended semantics of algebraic data types has often been formulated in terms of initial algebras, where both initiality and algebras are interpreted in the sense of category theory (Goguen et al. 1977).[4] We shall now show the compatibility of the constructive standpoint of formal systems with the algebraic and categorical language. We will do this by expressing constructions on formal systems in a notation that is immediately interpretable in categories, and will allow us to imitate the standard construction of the initial algebra (Adámek 1974).

[4] Some proviso is needed, however, on the correpondence between programming language constructs and logical and algebraic notions. For example, in languages with lazy pattern matching, like Haskell, the elements of type `Nat` are not in bijective correspondence with the natural numbers: in Haskell we can define `infty = Succ infty` for which the compiler infers type `Nat`, which does not correspond to any natural number but can nevertheless be used significantly as an argument of functions without causing non-terminating behavior (see (Bird 1998) for examples). Other expressions for which the type `Nat` can be inferred but which do not correspond to any natural number are introduced by defining `bottom = bottom` and then taking `bottom`, `Succ bottom`, `Succ (Succ bottom)`, ... The type `Nat` is more accurately modeled by a partially ordered set enjoying a special completeness property in the order-theoretic sense – a *cpo*; here, in addition to natural numbers, there is an infinite totally ordered subset whose elements corresponding to elements of `Nat` that involve `bottom`, whose least upper bound is the element corresponding to `infty`. This structure is still an initial algebra, but in a suitable category of cpo's (Freyd 1991).

3.1 Processes and their products

Given the data that specify an inductive class $\mathfrak{X}$, it is possible to define the notion of a *construction* of an element of $\mathfrak{X}$. According to Curry, "the operations are regarded as steps of construction, which may be iterated indefinitely, for forming new obs" (Curry 1952, p. 252, fn. 2) so that "an operation is [...] regarded as forming a new object rather than as assigning a value" (ibid.). The ob b formed by ω from $a_1, \ldots, a_n$ is denoted by $\omega(a_1, \ldots, a_n)$ but here Curry adds (ibid., p. 253):

> the genesis of such a b from $\omega, a_1, \ldots, a_n$ will also be referred to as a *formation* (of b), and for this formation ω will be called the *operation*, $a_1, \ldots, a_n$ the *arguments*, and b the *closure*.

Now, this is the only place in Curry's writings where this distinction is made. Yet, separating the process of ob formation from the resulting ob is a natural idea, and is directly related to the process/product distinction on which there is a significant philosophical literature, for instance (Twardowski 1999), see (Bobryk 2009; van der Schaar 2013).[5] Like drawing inferences within a formal system, steps of formation of obs are *actions*. For this reason, the study of general formal systems needs a systematic development of a terminology and techniques for dealing with these dynamic aspects. That is, we need a theory of processes and their results. While no attempt at building such a theory will be made in the present paper, we will however take up Curry's remark and distinguish between:

- the formation of an ob $b = \omega(a_1, \ldots, a_n)$ (*process*), denoted by

$$\omega \cdot \langle a_1, \ldots, a_n \rangle$$

- the closure $\overline{\xi}$ of a formation ξ (*product*).

We define, in general:

$$\omega(a_1, \ldots, a_n) \equiv \overline{\omega \cdot \langle a_1, \ldots, a_n \rangle}$$

We use $a :: A$, when A is a formal system, to express the judgement that a is an ob of A. Given a class of operations (a *signature*) Ω (where Ω_n is the subset of operations of degree n) we can define simultaneously formal systems Ω^* and $\Omega(X)$ for any formal system X, looking at them as types:

- $\Omega(X)$ is the type of *formations* with operation in Ω and obs of X as arguments,
- Ω^* is the type of *obs* built from operations in Ω

with the following rules, where the notation exploiting '::' is used to express informally the new elementary statements:

[5] Observe that also the terms 'closure', 'formation' are ambiguous and may refer both to processes and to their results, exactly like 'construction'.

$$\frac{\omega \in \Omega_n \quad x_1 :: X, \ldots, x_n :: X}{\omega \cdot \langle x_1, \ldots, x_n \rangle :: \Omega(X)} \text{ (formation)}$$
$$\frac{\omega \cdot \langle x_1, \ldots, x_n \rangle :: \Omega(\Omega^*)}{\omega(x_1, \ldots, x_n) :: \Omega^*} \text{ (closure)}$$

3.2 Constructing free structures

Now, let A and B be formal systems. We can define a notion of constructive transformation f from obs of A to obs of B. We define therefore the notation

$$f : A \longrightarrow B$$

to mean that f is an *effective process*, in the sense of (Curry 1963, §2A5), that transforms obs $a :: A$ into obs $f(a) :: B$. As a first example, we have a process

$$\delta : \Omega(\Omega^*) \longrightarrow \Omega^* \tag{2}$$

where, for an operation ω of degree n and $x_1, \ldots, x_n :: \Omega^*$, we form the ob $\omega(x_1, \ldots, x_n) :: \Omega^*$ as the closure of the formation $\omega \cdot \langle x_1, \ldots, x_n \rangle :: \Omega(\Omega^*)$, as in the derivation

$$\frac{\dfrac{\omega \in \Omega_n \quad x_1 :: \Omega^* \cdots x_n :: \Omega^*}{\omega \cdot \langle x_1, \ldots, x_n \rangle :: \Omega(\Omega^*)} \text{ (formation)}}{\omega(x_1, \ldots, x_n) :: \Omega^*} \text{ (closure)}$$

We can extend the construction of $\Omega(\cdot)$ to any $f : A \longrightarrow B$, obtaining a process

$$\Omega(f) : \Omega(A) \longrightarrow \Omega(B)$$

which is defined by the following specifications:

- $\Omega(f)$ takes an ob $\omega \cdot \langle a_1, \ldots, a_n \rangle :: \Omega(A)$ as argument;
- performs the process f on each of the arguments $a_1, \ldots, a_n$, producing eventually obs $f(a_1), \ldots, f(a_n)$ of B (by the effectiveness of f, these exist);
- produces $\omega \cdot \langle f(a_1), \ldots, f(a_n) \rangle :: \Omega(B)$.

Let now $\emptyset$ be the empty formal system: no rules, no theorems. The obs of the formal system $\Omega(\emptyset)$ correspond to the formations with operations of degree 0, if any. Then the obs of the formal system $\Omega(\Omega(\emptyset))$ are the formations whose arguments are obs in $\Omega(\emptyset)$, and this process can be iterated indefinitely. There is an obvious effective process $\Box : \emptyset \longrightarrow \Omega^*$ ("do nothing"), that can be taken as the starting point of an iterative construction whose stages are the processes:

$$
\begin{aligned}
i_0 = &\quad \emptyset \xrightarrow{\Box} \Omega^* \\
i_1 = &\quad \Omega(\emptyset) \xrightarrow{\Omega(i_0)} \Omega(\Omega^*) \xrightarrow{\delta} \Omega^* \\
&\quad \vdots \\
i_n = &\quad \Omega(\Omega^{n-1}(\emptyset)) \xrightarrow{\Omega(i_{n-1})} \Omega(\Omega^*) \xrightarrow{\delta} \Omega^* \\
&\quad \vdots
\end{aligned}
$$

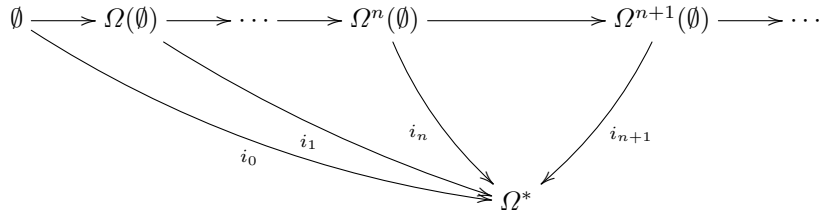
At each stage we add one layer of operations to the formations obtained cumulatively at earlier stages. So we form constants, which are operations of degree 0 and do not need any argument, then we have formations whose arguments are formations of constants and so on, indefinitely. At each stage, we can form new obs in Ω^* by taking the closures of the formations belonging to that stage, applied to the obs obtained at the earlier stages by means of applications of $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$ performed from the inside out. Thus these processes represent the *constructions* of increasing height of elements of Ω^*, defined in (Curry 1963, §2A6). Assume for instance that Ω consists of the three operations $\{\omega_2, \sigma_1, \gamma_0\}$, with subscripts indicating the degrees; we have the following formations:

$$
\begin{aligned}
\gamma_0 \cdot \langle \quad \rangle &:: \Omega(\emptyset) \\
\sigma_1 \cdot \langle \gamma_0 \cdot \langle \quad \rangle \rangle &:: \Omega(\Omega(\emptyset)) \\
\omega_2 \cdot \langle \gamma_0 \cdot \langle \quad \rangle, \sigma_1 \cdot \langle \gamma_0 \cdot \langle \quad \rangle \rangle \rangle &:: \Omega(\Omega(\Omega(\emptyset)))
\end{aligned}
$$

Observe that for *any* $\gamma_0 \cdot \langle \quad \rangle :: \Omega(\emptyset)$ we have $\gamma_0 \cdot \langle \quad \rangle :: \Omega(X)$ for any formal system X, and therefore also $\gamma_0 \cdot \langle \quad \rangle :: \Omega(\Omega^*)$. We can now display the derivations in tree form, showing how obs are generated from the leaves of the derivation tree towards its root:

$$
\dfrac{\dfrac{\dfrac{}{\gamma_0 \cdot \langle \quad \rangle :: \Omega(\Omega^*)}\ \text{(formation)}}{\gamma_0 :: \Omega^*}\ \text{(closure)} \qquad \dfrac{\dfrac{\dfrac{}{\gamma_0 \cdot \langle \quad \rangle :: \Omega(\Omega^*)}\ \text{(formation)}}{\gamma_0 :: \Omega^*}\ \text{(closure)} \qquad \dfrac{\dfrac{}{\sigma_1 \cdot \langle \gamma_0 \rangle :: \Omega(\Omega^*)}\ \text{(formation)}}{\sigma_1(\gamma_0) :: \Omega^*}\ \text{(closure)}}{\omega_2 \cdot \langle \gamma_0, \sigma_1(\gamma_0) \rangle :: \Omega(\Omega^*)}\ \text{(formation)}}{\omega_2(\gamma_0, \sigma_1(\gamma_0)) :: \Omega^*}\ \text{(closure)}
$$

We can visualize the processes involved in a familiar diagram that displays the construction of Ω^* as the term algebra over the signature Ω:



(3)

In our constructive reading, the structure Ω^* is in a sense the smallest possible that is closed under the process $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$. If we have another process

$$\xi : \Omega(X) \longrightarrow X$$

we can build an effective transformation $h : \Omega^* \longrightarrow X$ by the following recursive specification. An ob $a :: \Omega^*$ of the form $\omega(a_1, \ldots, a_n)$ is the closure of a unique formation in $\Omega(\Omega^*)$, where the arguments $a_1, \ldots, a_n$ have a strictly smaller height than a. Assuming that the process h is already defined for the arguments, we can assume that $h(a_1), \ldots, h(a_n)$ have been obtained as obs of X by the assumption on the effectiveness of h. Therefore

$$\omega \cdot \langle h(a_1), \ldots, h(a_n) \rangle :: \Omega(X)$$

so we can perform ξ on this ob producing an ob $\xi(\omega \cdot \langle h(a_1), \ldots, h(a_n) \rangle)$ of X as a result. This can be taken as the definition of the result $h(a)$ of performing $h : \Omega^* \longrightarrow X$ on $a :: \Omega^*$.

This construction of Ω^* is immediately interpretable in the more general context of categories, functors and the very general notion of algebra of a functor. Given a category $\mathscr{C}$ and a functor $T : \mathscr{C} \longrightarrow \mathscr{C}$, a *$T$-algebra* is an object C of $\mathscr{C}$ with a morphism $c : T(C) \longrightarrow C$. The reason for this name is that, if Ω is a signature, we can identify it with a functor $\Omega : \mathbf{Set} \longrightarrow \mathbf{Set}$ by setting

$$\Omega(X) = \{\langle \omega, \langle x_1, \ldots, x_n \rangle \rangle \mid \omega \text{ of degree } n \text{ and } x_1, \ldots, x_n \in X\}.$$

Then an Ω-algebra can be seen to correspond to a set X with an assignment of an n-ary operation over X to every $\omega \in \Omega$ of degree n, which is the standard notions of algebra for a signature Ω.

If $c : T(C) \longrightarrow C$ and $d : T(D) \longrightarrow D$ are T-algebras, a T-algebra homomorphism between them is defined to be a morphism $h : C \longrightarrow D$ of $\mathscr{C}$ that makes the following diagram commute:

$$\begin{array}{ccc} T(C) & \xrightarrow{c} & C \\ {\scriptstyle T(h)}\downarrow & & \downarrow{\scriptstyle h} \\ T(D) & \xrightarrow{d} & D \end{array}$$

An initial T-algebra is then one for which there is exactly one T-algebra homomorphism to every other T-algebra. It is the initial object of the category of T-algebras and their homomorphisms. As a very important example, Ω^* together with $\delta : \Omega(\Omega^*) \longrightarrow \Omega^*$, both intepreted as relative to the category **Set**, is the initial Ω-algebra. There are two basic facts about algebras in this sense that make them central in the categorical account of algebraic data types:

- there is a fundamental result by Lambek (1968), the *Lambek Lemma*, which states that if $a : T(A) \longrightarrow A$ is initial, then a is an isomorphism, that is, A is a fixed-point of the functor $T : \mathscr{C} \longrightarrow \mathscr{C}$. This guarantees the existence of solutions to recursive equations like `data Nat = Zero | Succ Nat` in the code shown in Figure 1: just take $\texttt{Nat} = \Omega^*$ for $\Omega = \{\texttt{Zero}, \texttt{Succ}\}$.
- Under very general assumptions on T and $\mathscr{C}$, in particular the existence of suitable colimits (Adámek 1974), the initial algebra $a : T(A) \longrightarrow A$ of T can be built as the colimit of a chain starting with the initial object, much like that of diagram (3).

From the point of view of programming, the universal property of initial algebras can be immediately turned into a scheme of definition by structural recursion of functions over data-types, when the latter are regarded as initial algebras. The explicit observation that many useful functions over data-types can be programmed as homomorphisms by means of such structural recursion schemes goes back at least to Burstall and Landin (1969), who extended this remark also to other free structures, in particular lists of elements of X regarded as elements of the free monoid over X, defining the Haskell functions `map`, `filter`, `any`, `all`, and substitution over $W_\Omega(x)$ as suitable homomorphisms from free structures, possibly over sets of generators.[6]

4 Iteration and inversion

4.1 On the essence and meaning of `Nat`

We have seen that it is possible to look at Ω as a functor from the category of sets to itself. Then we can interpret diagram (3) as a colimit, which implies that Ω^* is the (carrier of the) *initial* Ω-algebra

$$\delta : \Omega(\Omega^*) \longrightarrow \Omega^*.$$

[6] The interest of T-algebras in a computational setting can also be seen from their use in the categorical investigations on classes of automata by Arbib, Manes and several others, in the early 1970s. There, a central notion is that of *dynamics* that generalizes the transition function of an automaton $\delta : X \times Q \longrightarrow Q$, where X is the input alphabet and Q the set of states of the automaton. The observation that the construction $X \times \cdot$ is an endofunctor over the category of sets makes this notion of dynamics a special case of the general categorical definition of an algebra of an endofunctor $T : \mathscr{C} \longrightarrow \mathscr{C}$. In the context of the categorical reconstruction of automata theory, T-algebras were usually studied through the *free* monad over T, see (Arbib and Manes 1974) for an early survey of this field. Monads have come to play an important role in structuring Haskell programs, although through a different path, following pioneering work by Moggi, Spivey and Wadler in the late 1980s.

We have thus recovered the usual categorical notion of initiality that has been taken as essential to the characterization of data types in programming since the work of (Goguen et al. 1977).

This notion actually arose well before the invention of categories, in a way that can however immediately be expressed in categorical terms, in the classic account of Richard Dedekind (1888). This is the "Theorem of the definition by induction", which states that given a set A, an element $a \in A$ and a function $f : A \longrightarrow A$, there exists a unique $h : \mathbb{N} \longrightarrow A$ such that

$$h(0) = a$$
$$h(s(y)) = f(h(y)).$$

As the function $h : \mathbb{N} \to A$ depends uniformly both on $a \in A$ and on $f : A \longrightarrow A$, we can highlight this uniformity by writing it as foldn f a, as an interpretation of the `foldn` functional over `Nat` defined in Figure 1.

Fold functionals in programming have been used, for lists, by Barron and Strachey in 1963 (Danvy and Spivey 2007), but a particular case with the name of *reduction* was systematically present in APL (Iverson 1962, §1.8) in examples like the definition of multiplication as repeated addition:

$$+/5\ \rho\ 6 = 6 + 6 + 6 + 6 + 6 = 30$$

where $M\ \rho\ N$ yields the vector consisting of the M-fold repetition of N, and the sum reduction $+/$ performs the sum of all the elements of this vector. But the algebraic foundations of fold, and even their categorical connections, become explicit only in the algebra of programs over lists developed by Bird (1987) and in the Bird-Meertens formalism (or *Squiggol*) (Bird and Meertens 1987).

The essential use of folding in defining functions over the elements of algebraic data types can be understood better if we consider the second-order definitions of such structures (Böhm and Berarducci 1985). In the case of natural numbers, for example, we represent n as the second-order typed term

$$\Lambda X(\lambda f^{X\to X}.\lambda x^{X}.\underbrace{f(\cdots f}_{n\ \text{times}}\ x))$$

of type $\forall X.(X \to X) \to (X \to X)$, where X is a (universally quantified) type variable. This type is the standard encoding of $\mathbb{N}$ in this context. Given any $\langle A, s, z\rangle$ such that $z : A$ and $s : A \to A$ there is an element $n\ A\ s\ z : A$, representing the result of iterating n times the "successor" function s to the "zero" element z. Here type abstraction, expressed by the prefix ΛX, allows to perform the iteration of $s : A \to A$ starting from $z : A$, over *any* carrier A and for *any* choice of $s : A \to A$ and $z : A$, and acts as a surrogate of the initiality of $\mathbb{N}$.

This way of characterizing the natural numbers as iterators can already be found literally in Wittgenstein's *Tractatus* (Frascolla 1997), and can be grafted onto the tradition of proof-theoretic semantics (Schroeder-Heister 2018) by exploiting a form of the inversion principle of Lorenzen, relating introduction and

elimination rules. It is this set of notions that we would like now to introduce briefly.

4.2 Folding and inversion

An important idea, formulated by Gentzen (1935) for his systems of natural deduction, is that the elimination rules for logical constants are in some sense consequences of the corresponding introduction rules (Curry 1963, §5A3, p. 173). This idea can be made more precise by an application of the *inversion principle* of Lorenzen (1955),[7] formulated originally for his notion of *calculus* but readily adapted to the context of formal systems.

We first need an important notion in the general theory of formal systems, that of *admissibility* of a rule of inference. A rule R is *admissible* if any derivation obtained by the use of R can be transformed into one obtained without such use, see (Curry 1963, §3A2). In the formal system N for natural numbers introduced in (1), the rule

$$\frac{N(s(x))}{N(x)}$$

is admissible, though it cannot be obtained by composing the rules of the system. The proof of its admissibility proceeds by induction on the height of the derivation of the premise, which in the present case could only have been derived by an application of rule

$$\frac{N(x)}{N(s(x))}\ \text{(successor)}$$

so $N(x)$ must already have been derived. Observe that the same argument allows to define a process $p : N \longrightarrow N$ that computes the predecessor function over positive numbers.

Now, the *inversion principle* states that if, in a formal system, A can only be introduced by the rules

$$\frac{\Gamma_1}{A} \cdots \frac{\Gamma_n}{A}$$

then the rule

$$\frac{A}{C}$$

is admissible when

$$\frac{\Gamma_1}{C} \cdots \frac{\Gamma_n}{C}$$

are used as additional rules. Indeed, given a derivation of A, its last step must be of one of the forms

$$\frac{\Gamma_i}{A}$$

[7] See (Schroeder-Heister 2008; Moriconi and Tesconi 2008) for recent investigations.

for some $i = 1, \ldots n$. Then an application of rule

$$\frac{\Gamma_i}{C}$$

allows to infer C, showing admissibility.

This reasoning can be generalized to the situation where we have a formal system A, with an ob $a :: A$ and an effective process $f : A \longrightarrow A$. Consider then the possible ways in which we can "introduce" in a derivation of N a judgement of the form $N(n)$:

1. the judgement is $N(0)$ and follows from axiom (zero);
2. the judgement has the shape $N(s(x))$ and is derived by an application of rule (successor).

and we can define an effective "elimination" process $F(f, a) : N \longrightarrow A$ by induction on the height of the derivation of the judgement $N(n)$ for each $n :: N$:

– in case 1, $n \equiv 0$ and

$$F(f, a)(0) = a \tag{4}$$

– in case 2, $n \equiv s(x)$ and

$$F(f, a)(s(x)) = f(F(f, a)(x)). \tag{5}$$

The process $F(f, a)$ has the same definition as foldn f a, and relates this generalization of the inversion principle to the universal property of free structures, in this case N. Equations (4) and (5) define folding as elimination rule for N, where the obs of N become iterators and the constructors 0 and s play the role of introduction rules.[8]

5 A confluence of languages

By way of conclusion, we can now put our story in a nutshell. Starting from some basic ideas originating from Curry's work on formal systems, we have reached a point where the languages of algebra, proof theory and category theory can all be used to describe a set of notions exploited by an extensive style of functional programming, illustrated by the Haskell code displayed in Figure 1. Universal properties of structures yield recursion schemes defining unique homomorphisms that characterize data structures whose elements can be synthesized or analyzed by means of proof-theoretical notions like introduction/elimination rules, related by the inversion principle. All the notions we have encountered along our story converge and hint at possible common foundations. In this paper we have made an attempt to argue that the existence of such a convergence can be justified on a conceptual basis.

[8] See (Thompson 1991) for more on the application of the inversion principle to programming.

One possible development of the present work should extend the kind of analysis sketched in this paper to the potentially infinite structures that have come to play a prominent role in (lazy) functional programming (Meijer et al. 1991). The semantical foundations of these structures have been thoroughly investigated both with the techniques of domain theory and with those arising from the theory of final coalgebras and coinduction (Fiore 1996). Yet, we believe that it may be useful to present them in a coherent narrative showing, also in this case, the confluence of notions and accounts couched in different languages.

From a more general point of view, we believe that the overall structure of the story presented in this paper, with its focus on a relevant fragment of code and the development of its conceptual context, may have many other instances within a programme systematically investigating the constitution of programming paradigms by following, as we have attempted to do here, the formation of sets of linguistic abstractions from different and often unrelated insights into computational problems.

Acknowledgements The preparation of this paper has been supported by project PROGRAMme ANR-17-CE38-0003-01 (principal investigator Liesbeth De Mol). I am grateful to the anonymous referees for insightful comments that have led to a definite improvement of the original version. My warmest thanks also to Simone Martini for presenting the results of this paper at a project meeting that I could not attend.

References

Adámek J (1974) Free algebras and automata realizations in the language of categories. Commentationes Mathematicae Universitatis Carolinae 15(4):589–602

Arbib MA, Manes EG (1974) Machines in a category: An expository introduction. SIAM Review 16(2):163–192

Bauer FL (1990) The cellar principle of state transition and storage allocation. IEEE Annals of the History of Computing 12(1):41–49

Bauer FL (2002) From the Stack Principle to ALGOL. In: Broy M, Denert E (eds) Software Pioneers, Springer-Verlag, pp 26–42

Bird RS (1987) An introduction to the theory of lists. In: Broy M (ed) Logic of Programming and Calculi of Discrete Design, Springer-Verlag, pp 3–42, NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University

Bird RS (1998) Introduction to Functional Programming Using Haskell. Prentice-Hall

Bird RS, Meertens L (1987) Two exercises found in a book on algorithmics. In: Meertens L (ed) Program Specification and Transformation, North-Holland, pp 451–457

Bobryk J (2009) The genesis and history of Twardowski's theory of actions and products. In: Lapointe S, Wolénski J, Marion M, Miskiewicz W (eds) The

Golden Age of Polish Philosophy: Kazimierz Twardowski's Philosophical Legacy, Springer Netherlands, Dordrecht, pp 33–42
Böhm C, Berarducci A (1985) Automatic synthesis of typed λ-programs on term algebras. Theoretical Computer Science 39:135–154
Burks AW, Warren DW, Wright JB (1954) An analysis of a logical machine using parenthesis-free notation. Mathematical Tables and Other Aids to Computation 8:53–57
Burstall RM (1969) Proving properties of programs by structural induction. The Computer Journal 12(1):41–48
Burstall RM, Landin PJ (1969) Programs and their proofs: An algebraic approach. In: Meltzer B, Michie D (eds) Machine Intelligence, Edinburgh University Press, vol 4, pp 17–43
Cohn P (1965) Universal Algebra. Harper's series in modern mathematics, Harper & Row
Coquand T, Huet G (1988) The calculus of constructions. Information and Computation 76:95–120
Curry HB (1934) Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the USA 20:584–590
Curry HB (1941) Some aspects of the problem of mathematical rigor. Bulletin of the American Mathematical Society 47:221–241
Curry HB (1950) Language, metalanguage, and formal systems. Philosophical Review 59:346–353
Curry HB (1951) Outlines of a Formalist Philosophy of Mathematics. North-Holland Co., Amsterdam
Curry HB (1952) On the definition of substitution, replacement and allied notions in an abstract formal system. Revue Philosophique de Louvain 50:251–269
Curry HB (1953) Mathematics, syntactics and logic. Mind 62:172–183
Curry HB (1961) Some logical aspects of grammatical structure. In: Jakobson R (ed) Structure of Language and its Mathematical Aspects, no. 12 in Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, R.I., U.S.A., pp 56–68
Curry HB (1963) Foundations of Mathematical Logic. McGraw-Hill, New York, reprinted 1977 by Dover, Inc., New York.
Curry HB, Feys R (1958) Combinatory Logic, Volume I. North-Holland Co., Amsterdam, (3rd edn. 1974)
Curry HB, Hindley JR, Seldin JP (1972) Combinatory Logic, Volume II. North-Holland Co., Amsterdam
Danvy O, Spivey M (2007) On Barron and Strachey's Cartesian Product Function. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP '07, pp 41–46
De Mol L (2006) Closing the circle: an analysis of Emil Post's early work. Bulletin of Symbolic Logic 12(2):267–289
Dedekind R (1888) Was sind und was sollen die Zahlen?, 1st edn. Verlag von Friedrich Vieweg und Sohn, Braunschweig, translation by W.W. Beman in *Essays on the Theory of Numbers* (1901), reprinted in 1963 by Dover Press

Fiore MP (1996) A coinduction principle for recursive data types based on bisimulation. Information and Computation 127(2):186 – 198

Frascolla P (1997) The Tractatus system of arithmetic. Synthese 112(3):353–378

Freyd PJ (1991) Algebraically complete categories. In: Carboni A, Pedicchio M, Rosolini G (eds) Proceedings of the 1990 Como Category Theory Conference, Springer-Verlag, Lecture Notes in Mathematics, vol 1488, pp 131–156

Gentzen G (1935) Untersuchungen über das logische Schliessen. Mathematische Zeitschrift 39:176–210, 405–431

Goguen J, Thatcher J, Wagner E, Wright J (1977) Initial algebra semantics and continuous algebras. Journal of the ACM 24:68–95

Gorn S (1959) Introductory speech. In: Proceedings of the International Conference on Information Processing, UNESCO, Paris, June 13–22 1959, pp 117–118

Hindley JR (1969) The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society 146:29–60

Hoare CAR (1972) Notes on data structuring. In: Dahl OJ, Dijkstra EW, Hoare CAR (eds) Structured Programming, Academic Press Ltd., London, UK, pp 83–174

Hudak P, Hughes J, Peyton Jones SL, Wadler P (2007) A history of Haskell: being lazy with class. In: Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007, pp 1–55

Iverson KE (1962) A Programming Language. John Wiley & Sons, Inc., New York, NY, USA

Lambek J (1968) A fixpoint theorem for complete categories. Mathematische Zeitschrift 103:151–161

Landin PJ (1964) The mechanical evaluation of expressions. The Computer Journal 6:308–320

Lorenzen P (1955) Einführung in die Operative Logik und Mathematik. Springer-Verlag, Berlin, Göttingen, Heidelberg

Martin-Löf P (1982) Constructive mathematics and computer programming. In: Cohen LJ, Łos J, Pfeiffer H, Podewski KP (eds) Logic, Methodology and Philosophy of Science, VI, North-Holland Co., Amsterdam, pp 153–175

McCarthy J (1963) Towards a mathematical science of computation. In: Popplewell CM (ed) Information Processing 62: Proceedings of the IFIP Congress 1962, North-Holland Co., Amsterdam, pp 21–28

Meijer E, Fokkinga MM, Paterson R (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In: Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings, pp 124–144

Meyer RK (1987) Curry's philosophy of formal systems. Australasian Journal of Philosophy 65(2):156–171

Milner R (1978) A theory of type polymorphism in programming. Journal of Computer and System Sciences 17:348–375

Moriconi E, Tesconi L (2008) On inversion principles. History and Philosophy of Logic 29(2):103–113

Post EL (1943) Formal reductions of the general combinatorial decision problem. American Journal of Mathematics 65(2):197–215

Rosenbloom P (1950) The Elements of Mathematical Logic. Dover Inc., New York

van der Schaar M (2013) On the ambiguities of the term *judgement*; an evaluation of Twardowski's distinction between action and product. In: Chrudzimski A, Lukasiewicz D (eds) Actions, Products, and Things. Brentano and Polish Philosophy, De Gruyter, Berlin and Boston, pp 35–54

Schroeder-Heister P (2008) Lorenzen's operative justification of intuitionistic logic. In: van Atten M, Boldini P, Bourdeau M, Heinzmann G (eds) One Hundred Years of Intuitionism (1907–2007): The Cerisy Conference, Birkhäuser, pp 214–240

Schroeder-Heister P (2018) Proof-theoretic semantics. In: Zalta EN (ed) The Stanford Encyclopedia of Philosophy, spring 2018 edn, Metaphysics Research Lab, Stanford University

Seldin JP (1975) Arithmetic as a study of formal systems. Notre Dame Journal of Formal Logic 16(4):449–464

Thompson S (1991) Type Theory and Functional Programming. Addison Wesley Longman Publishing Co., Inc.

Twardowski K (1999) Actions and products (1912). In: Brandl J, Wolenski J (eds) On Actions, Products and Other Topics in Philosophy, Rodopi, Amsterdam, pp 103–132

Wozencraft JM, Evans A Jr (1971) Notes on programming linguistics. Tech. rep., Massachusetts Institute of Technology, Cambridge, Massachusetts